Toupie User's Manual

Marc-Michel Corsini and Antoine Rauzy

Contents

1 Introduction **2**

2 How To Run Toupie **3**
- 2.1 Getting Started . . . 3
- 2.2 Reading Programs . . . 3
- 2.3 Inserting Definitions at the Terminal . . . 4
- 2.4 Directives: Queries and Commands . . . 4
- 2.5 Syntax Errors . . . 4
- 2.6 Exiting from the Interperter . . . 4
- 2.7 Saving and Restoring a Program . . . 4

3 Commands **5**
- 3.1 Commands . . . 5
- 3.2 Display Commands . . . 6

4 The Toupie Language **8**
- 4.1 Syntax . . . 8
  - 4.1.1 Terms . . . 8
  - 4.1.2 Formulae . . . 9
  - 4.1.3 Fixpoint Definitions . . . 9
  - 4.1.4 Additional Features . . . 9
- 4.2 Semantics . . . 10

5 Programming Examples **13**
- 5.1 The Wolf, the Goat and the Cabbage . . . 13
- 5.2 Nim Game . . . 14
- 5.3 Milner's scheduler . . . 16

A Syntax **20**

B Reported Bugs **22**

Chapter 1

Introduction

Toupie is a μ-calculus interpreter in which variables take their values in (symbolic) finite domains. Toupie uses decision diagrams (an extension to finite domains of Bryant's Binary Decision Diagrams [Bry86, BRB90, Bry92]) to encode formulae and relations. Initially, Toupie was designed to perform abstract interpretation of Prolog programs [CMR92]. The current version has a wide spectrum of applications, including model checking [CGR93], static analysis of declarative languages [CCMR93], and resolution of classical Artificial Intelligence puzzles.

This manual describes the version 0.24 of Toupie, developed at the Laboratoire Bordelais de Recherches en Informatique (LaBRI). The system is written in standard ANSI C and can thus run on many platforms (SUN Spark Station, IBM RISC 6000, PC linux, Apple Macintosh, Next Station), provided that there exist minimal ressources such as a video screen with *termcap* entry.

The user interface of Toupie is a small command interpreter. These commands permit the loading, observation and deletion of programs; as well as interactive queries and responses.

From a more theoretical point of view, there are many papers devoted to the (propositional) μ-calculus. But, since the syntax and the semantics we have adopted do not follow these usual presentations, we have tried to make this manual as self-contained as possible.

Remarks, comments are welcome:

Antoine Rauzy
LaBRI – URA CNRS 1304 – Université Bordeaux I
33405 Talence cedex – FRANCE
tel: (33) 56 84 60 83, fax: (33) 56 84 66 69
e-mail: rauzy@labri.u-bordeaux.fr

Chapter 2

How To Run Toupie

Toupie offers the user an interactive programming environment for incrementally building programs.

The text of a Toupie program is normally created in a file (or a number of files) using a standard text editor. The Toupie interpreter can be instructed to read programs from these files.

2.1 Getting Started

Toupie is typically started from one of the UNIX shells by entering the command

```
% toupie
```

The interpreter responds with a banner message and the prompt '`2p |=`' as soon as it is ready to accept input; thus:

```
--- Toupie version 0.24 (04/10/93)
    Antoine Rauzy
    LaBRI - URA CNRS 1304 - Universite Bordeaux I
    351, cours de la Liberation - 33405 Talence - France
    rauzy@labri.u-bordeaux.fr

2p |=
```

2.2 Reading Programs

A program consists of a sequence of domain and fixpoint definitions, possibly interspersed with directives to the interpreter. The order between fixpoint definitions has no importance, but domains must be declared before their use.

To input a program from a file `file`, type the command:

```
2p |= load file
```

Note that it is not necessary to surround the file name with any additional symbols. Thus, the following command is legal:

```
2p |= load Benchmarks/Queens/queens8.2p
```

We use to adopt the extension ".2p" for files containing Toupie programs. Nevertheless, this extension is not implicitely added nor required.

Warning: in the current version, if a fixpoint equation is defined more than once, all versions are maintained. It may result an unexpected behavior.

2.3 Inserting Definitions at the Terminal

Domains and fixpoint definitions may also be typed directly at the terminal, although this is not recommended since the syntax is sufficiently complicated to induce many typographical errors. To enter a definition at the terminal, it suffices to type it. E.g. a domain definition:

```
2p |= let my_domain {a, b, c, d}
```

A fixpoint definition (predicate `p`):

```
2p |= p(X:my_domain,Y:my_domain) += (X#Y)
```

For the precise syntax of Toupie formulae and programs see Chap. 4.

2.4 Directives: Queries and Commands

Directives are either *queries* or *commands*. The full syntax of a query is a formula followed by a question mark `?`, e.g.

```
2p |= (X:{a,b,c} = Y:{d,c,b}) ?
```

This query contains two variables X and Y that belong respectively to the sets of symbolic constants $\{a, b, c\}$ and $\{d, c, b\}$. It looks for the pairs $< v, w >$ of possible values of X and Y, such that $v = w$.

If the query specified admits solutions (as in the previous example), the system answers:

```
{X=b, Y=b}
{X=c, Y=c}
```

meaning that there are two pairs of values that verify the equality. The execution of the query then terminates.

If all the possible assignments of the variables occurring in the query are solutions (e.g. `(X:{a,b}=X) ?`), the system answers `1`. If none of these assignments is solution (e.g. `(X:1..3#X) ?`) the system answers `0`.

2.5 Syntax Errors

Syntax errors are detected during reading. If an error occurs when reading a definition or a directive, a message is printed indicating at which line the error has been detected. E.g.

```
2p |= (X=Y) ?

2p ->  line 1: the variable X hasn't a domain.
```

Normally, no definition is stored and you can return to the interpreter by typing RETURN twice.

2.6 Exiting from the Interperter

To exit from the interpreter and return to the shell level, type a `^C` or a `^D` (UNIX interrupts), or use the command `exit`.

2.7 Saving and Restoring a Program

In order to suppress all of the definitions that are currently stored, use the command `clear all`. These definitions can be saved into a text file by means of the command `save` followed by a file name (as in the command `load`).

Chapter 3

Commands

This chapter describes the commands of the Toupie interface.

3.1 Commands

clear This command has two options: **clear program** which reinitializes the program (all the definitions are deleted); and **clear all** which reinitializes both the program and the decision diagrams manager.

dependencies Compute the dependency graph of each fixpoint definition, i.e. the links between the fixpoint definitions. These dependencies can be displayed (see Section 3.2).

display (see Section 3.2).

domain used to define a default domain. E.g.

```
2p |= domain {a, b, c}
```

Thereafter, if the domain of a variable is not specified at its first appearance, it is set by default to $\{a, b, c\}$.

let Followed by an identifier and a domain, defines a domain of the given name.

```
2p |= let a_name { a, b, c }
2p |= let another_name 1..4
```

Thereafter, it is possible to declare variables that belong to domain by giving its name. E.g.

```
2p |= (X:a_name = Y:another_name) ?
```

For implementation reasons, it is far more efficient to use named domains rather to repeat them when several variables have the same domain. The syntax of domain names is the same as the syntax of symbolic constants (and thus it is highly discouraged to use the same identifier as a constant and a domain name). Note that in `let name domain`, domain cannot be the name of a previously defined domain, and that it is not possible to redefine a named domain.

exit Terminates the Toupie cession and returns to control monitor.

garbage The Decision Diagrams garbage collector is called. Normally, it is not necessary for the user to call this command, since the garbage collection is automatically performed when there is not enough space available.

load followed by a file name `file`: inputs a program from the given file `file`. E.g.

```
2p |= load file
```

Note that it is not necessary to surround the file name with any delimiters. Thus, the following command is legal:

```
2p |= load Benchmarks/Queens/queens8.2p
```

We use to adopt the extension ".2p" for files containing Toupie programs. Nevertheless, this extension is not implicitely added nor required.

save Followed with a file name (with the same syntax that for the `load` command), saves the current program (i.e. domains and fixpoint definitions). E.g.

```
2p |= save Benchmarks/Queens/queens8.2p
```

stat Turn on the statistic option. As the result (in this version), the running time is printed after each request computation.

nostat Turn off the statistic option (this is the default value).

spy This command takes two arguments: a predicate name and its arity, i.e. a positive integer. The spied predicate is traced, that is a message is printed each time its value is recomputed. E.g.

```
2p |= spy p 2

2p -> predicate p/2 spied
```

nospy This command has the same syntax as **spy**. It disactives the spy option for the given predicate.

system A UNIX command is read on the current line, and is then executed.

3.2 Display Commands

The display command takes an argument:

domain Displays the current default domain. E.g.

```
2p |= domain {a,b,c}

2p |= display domain

domain {a,b,c}
```

domains Displays all of the named domains together with their values. E.g.

```
2p |= let a_domain {a,b,c}

2p |= let b_domain 1..4

2p |= display domains

let a_domain {a,b,c}

let b_domain 1..4
```

equations Displays all of the fixpoint equations.

```
2p |= p(X:{a,b,c},Y:{d,c,b}) += (X=Y)

2p |= display equations

p(X@1:{a,b,c},Y@2:{d,c,b}) += (X=Y)
```

dependencies Displays, for each fixpoint equation, the predicates that must be recomputed when the defined one changes of value.

```
2p |= p(X:{a,b,c},Y:{d,c,b}) += (X=Y)

2p |= q(X:{a,b,c},Y:{d,c,b}) += (p(X,Y) | q(Y,X))

2p |= dependencies

2p |= display dependencies

p/2 --> q/2,
q/2 -->
```

memory For debugging purpose only: displays the current state of the decision diagrams manager.

Followed by an predicate identifier, **display** displays the fixpoint definitions of the predicates that have the given name.

```
2p |= p(X:{a,b,c},Y:{d,c,b}) += (X=Y)

2p |= display p

p(X@1:{a,b,c},Y@2:{d,c,b}) += (X=Y)
```

Chapter 4

The Toupie Language

This chapter provides a brief introduction to the syntax and semantics of Toupie programs.

4.1 Syntax

There are three syntactic categories in Toupie: *terms*, *formulae* and *predicate definitions*. A Toupie program is a set of predicate definitions having different head predicate symbols or distinct arities. A Toupie query is a formula.

4.1.1 Terms

The data objects of Toupie are called terms. A term is either a *constant*, a *domain* or a *variable*.

Constants:

The constant symbols are:

- The *positive integers* such as 0, 1, 999. Note that arithmetical operations are not supported in the current version.
- The *symbolic constants*, that is any identifier in $[a..z][a..zA..Z0..9_]^\star$ (e.g., `aConstant`, `the_9th_constant`, ...).

Domains:

A set of constants is called a *domain*. There are two kinds of domains:

- *Ranges*, written $i..j$, where i and j are positive integers such that $i < j$.
- *Sets*, written $\{k_1, \ldots k_r\}$, where the k_i's are constants.

It is possible to give a name to a domain, e.g.

```
2p |= let a_name { a, b, c }
```

For implementation reasons, it is far more efficient to use named domains, when several variables have the same domain. The syntax of domain names is the same than the syntax of symbolic constants (and thus it is highly recommended to do not use the same identifier as a constant and a domain name).

It might be the case that all the variables of a program belong to the same domain. A default domain can be defined, e.g.

```
2p |= domain {a, b, c}
```

Then, it is possible to declare a variable without specifying its domain; the variable then belongs to the default domain.

Variables:

The variable symbols are identifiers in $[A..Z][a..zA..Z0..9_]^\star$ (e.g., `The_9th_variable, Avariable`, ...).

Each variable occurring in a Toupie expression must have an interpretation domain. This domain must be declared with the first occurrence of the variable. A domain declaration for the variable X is of the form: `X : domain`, where `domain` is any domain, included a domain name. In this case, the domain must has been declared before.

Variables in different fixpoint definitions or requests are completely independent, even if they have the same name — the lexical scope of a variable is limited to a single fixpoint definition or request.

4.1.2 Formulae

Formulae have the following form:

- The two Boolean constants `0` and `1`.
- `(X1=X2)` or `(X1=k)` or `(X1#X2)` or `(X1#k)` where `X1` and `X2` are variables and `k` is a constant symbol.
- `(X in domain)` where X is a variable and `domain` is any domain, including a domain name. In this case, the domain must has been declared before.
- `P(X1,..., Xn)` where `P` is an n-ary predicate variable and `X1,...,Xn` are individual variables.
- `~ f, (f & g), (f | g), (f <=> g), (f <#> g), (f => g)` where `f` and `g` are formulae and `~, &, |, <=>, <#>, =>` denote the logical connectives $\neg, \wedge, \vee, \Longleftrightarrow, \otimes$ and $\Rightarrow$. Note that the parentheses are absolutely required.
- `@(i,[f1,...,fn])`, where `i` is positive integer and `f1 ... fn` are formulae. This connective is satisfied if and only if at least `i` fi's are satisfied.
- `#(i,j,[f1,...,fn])`, where `i` and `j` are positive integers and `f1...fn` are formulae. This connective is satisfied iff at least `i` and at most `j` fi's are satisfied.
- `forall X1,..., Xn f` or `exist X1,..., Xn f` where `X1,...,Xn` are variables and `f` is a formula. `forall` and `exist` denote respectively the two quantifiers $\forall$ and $\exists$.

4.1.3 Fixpoint Definitions

Predicate definitions are as follows:

- `P(X1,...,Xn) += f`
- `P(X1,...,Xn) -= f`

where `P` is an n-ary predicate variable, `X1,...,Xn` are individual variables, and `f` is a formula. The token `+=` and `-=` denote respectively least and greatest fixpoint definitions. The formula f is assumed to be monotonic. This property can been easily checked by a syntactic inspection of the formulae, but is NOT verified in the current version.

4.1.4 Additional Features

Comments

It is possible to have comments in a Toupie program. A `%` indicates that the text between this symbol and the end of the line is a comment.

Variable Ordering

The internal data structures of Toupie, namely the Decision Diagrams (for a more technical presentation of Toupie see [CR93a, CR93b]), require an order over the variables of a fixpoint definition or a formula. By default, the variables are ordered by their order of appearance.

It is often necessary, for sake of efficiency, to customize this order. An index can be thus declared with the first occurrence of the variable, with the following syntax: `X@i` where `i` is a positive integer. E.g.

```
let vertex 1..10

edge(X:vertex, Y:vertex) += (
  % relation that defines a graph
  ...
  )

closure(X@1:vertex,Y@3:vertex) += (
    edge(X,Y)
  | exist Z@2:vertex (edge(X,Z) & closure(Z,Y))
  )
```

If an unindexed variable occurs in a formula, its index is the integer that follows immediately the greatest defined index.

4.2 Semantics

The semantics of Toupie formulae is determined with respect to a structure $S = \langle Const, \mathcal{V} \rangle$ where $Const$ is an interpretation domain, and $\mathcal{V}$ is a denumerable set of variables including all the variables of the program. As in DATALOG, we assume, first, the hypothesis of unicity of names (two distinct constant symbols denotes two distinct constants), and second, the hypothesis of the closure of the domain, that is to say that $Const$ is the set of all the constants occurring in the considered formula or program.

Definition 1 [Individual Variable Assignments]
An *individual variable assignment* is a mapping α from $\mathcal{V}$ into $Const$ such that $\alpha(X) \in dom(X)$ for any variable X occurring in the program.

For sake of simplicity, we assume implicitely the condition $\alpha(X) \in dom(X)$.

Definition 2 [Relations]
A *relation* on S is a mapping from $\mathcal{V} \rightarrow Const$ into $\mathcal{B}$, where $X \rightarrow Y$ stands for the set of mappings from X to Y and $\mathcal{B}$ stands for the Boolean values.

Definition 3 [Predicate Variable Interpretations]
Let Pr be the set of predicates occurring in the program. A *predicate variable interpretation* is a mapping from Pr into $(\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$, where $\mathbb{N}$ stands for the set of natural numbers.

This definition avoids the complications due to the different arities of the predicates. For a predicate of arity n, it suffices to consider that the corresponding function depends only on the first n numbers.

The semantics of a formula is thus a relation, and the semantics of a predicate (defined with a fixpoint equation) is a mapping from $(\mathbb{N} \rightarrow Const)$ into $\mathcal{B}$.

A Toupie program P assigns a meaning to a set of predicate symbols Pr. The semantics of the program is defined as the least fixpoint of a transformation $\mathcal{T}$. Let us denote by $\mathcal{PR}$

the set $Pr \rightarrow (\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$ of predicate variable interpretations, and by $\mathcal{RE}$ the set $(\mathcal{V} \rightarrow Const) \rightarrow \mathcal{B}$ of relations.

The program defines a continuous transformation:

$$\mathcal{T} : \mathcal{PR} \rightarrow \mathcal{PR}$$

Each formula `f` defines a function:

$$\mathcal{T}[\![\texttt{f}]\!] : \mathcal{PR} \rightarrow \mathcal{RE}$$

And each equation `Eq` defines a function:

$$\mathcal{T}[\![\texttt{Eq}]\!] : \mathcal{PR} \rightarrow (\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$$

The definition of $\mathcal{T}$ will use the following notation.

Definition 4 [Substitutions]
Let $f : A \rightarrow B$ be a function. Let $a_1, \ldots, a_n$ be distinct elements of A, and $b_1, \ldots, b_n$ be arbitrary elements of B. We note by

$$f[a_1/b_1, \ldots, a_n/b_n]$$

the function $g : A \rightarrow B$ such that $ga_i = b_i$ $(1 \leq i \leq n)$ and $ga = fa$ $(\forall a \notin \{a_1, \ldots, a_n\})$.
The notation

$$[a_1/b_1, \ldots, a_n/b_n]$$

stands for $f[a_1/b_1, \ldots, a_n/b_n]$ where f is an arbitrary function.

We are now in position to define the semantic function $\mathcal{T}$. Let π be a predicate variable interpretation, α be an individual variable assignment, and σ be an element of $(\mathbb{N} \rightarrow Const)$. $\mathcal{T}$ is defined inductively on the structure of formulae in the following way :

- $\mathcal{T}[\![1]\!]\ \pi\ \alpha\ = 1$ and $\mathcal{T}[\![0]\!]\ \pi\ \alpha\ = 0$.
- $\mathcal{T}[\![X_i = X_j]\!]\ \pi\ \alpha\ = \alpha(X_i) = \alpha(X_j)$.
- $\mathcal{T}[\![X_i = k]\!]\ \pi\ \alpha\ = \alpha(X_i) = k$.
- $\mathcal{T}[\![\sim\ g]\!]\ \pi\ \alpha\ = \neg\mathcal{T}[\![f]\!]\ \pi\ \alpha$.
- $\mathcal{T}[\![f \mid g]\!]\ \pi\ \alpha\ = \mathcal{T}[\![f]\!]\ \pi\ \alpha \vee \mathcal{T}[\![g]\!]\ \pi\ \alpha$.
- $\mathcal{T}[\![f\ \&\ g]\!]\pi\alpha\ = \mathcal{T}[\![f]\!]\ \pi\ \alpha \wedge \mathcal{T}[\![g]\!]\ \pi\ \alpha$.
- $\mathcal{T}[\![\forall\ X\ f]\!]\ \pi\ \alpha\ = \bigwedge_{k \in dom(X)}(\mathcal{T}[\![f]\!]\ \pi\ \alpha[X/k])$.
- $\mathcal{T}[\![\exists\ X\ f]\!]\ \pi\ \alpha\ = \bigvee_{k \in dom(X)}(\mathcal{T}[\![f]\!]\ \pi\ \alpha[X/k])$.
- $\mathcal{T}[\![P(X_{i_1}, \ldots, X_{i_r})]\!]\ \pi\ \alpha = \pi(P)(\alpha(X_{i_1}), \ldots, \alpha(X_{i_r}))$.
- $\mathcal{T}[\![P(X_1, \ldots, X_n) += f\ ]\!]\ \pi\ \sigma = \mathcal{T}[\![f]\!]\ \pi\ [X_1/\sigma(1), \ldots, X_n/\sigma(n)]$.

Finally, the transformation associated with the program is:
$\mathcal{T}[\![Eq_1 \ldots Eq_n]\!]\ \pi = \pi[p_1/\mathcal{T}[\![Eq_1]\!]\ \pi, \ldots, p_n/\mathcal{T}[\![Eq_n]\!]\ \pi]$
where the p_i are the predicates defined by the equations Eq_i.

Definition 5 [Denotation of a Toupie Formula wrt. a Program]
Let P be a Toupie program. Let f be a Toupie formula. Let D be the set of free variables occurring in f. By definition, the *denotation* of f wrt. P is the function $\mathcal{D}[\![f]\!] : (D \rightarrow Const) \rightarrow \mathcal{B}$ such that, for all $\alpha \in (D \rightarrow Const)$,

$$\mathcal{D}[\![f]\!]\alpha\ =\ \mathcal{T}[\![f]\!](\mu(\mathcal{T}[\![P]\!]))\alpha',$$

where α' is any variable assignment such that

$$\alpha' X = \alpha X\ (\forall X \in D).$$

(The underlying program remains implicit.)

Note that the introduction of greatest fixpoint definitions complicates the notation but not the semantics itself.

Chapter 5

Programming Examples

In this chapter, three examples are examined in details.

5.1 The Wolf, the Goat and the Cabbage

The first problem we examine is the well-known boatman problem. The sad story is the following: a poor old boatman wants to cross the river with his wolf, his goat and cabbage; alas his two-place boat is too small for all of his belongings, the river is too deep to cross it by foot and the closest bridge is far. He cannot leave the wolf alone with the goat nor the goat alone with the cabbage.

To solve it, one could try the following program:

```
% the boatman story

domain {left,right}  % default domain declaration

% reachable describes all of the possible states, i.e. all the
% possible positions of the man (M), the wolf (W), the goat (G)
% and the cabbage (C).

reachable(M,W,G,C) += (
     % initial state: every one is on the left bank
     ((M=left) & (W=left) & (G=left) & (C=left))
   | (
        ( % the possible crossings
            (reachable(M2,W,G,C)  & (M2#M))                    % man alone
          | (reachable(M2,W2,G,C) & (M2=W2) & (M=W) & (M2#M)) % man + wolf
          | (reachable(M2,W,G2,C) & (M2=G2) & (M=G) & (M2#M)) % man + goat
          | (reachable(M2,W,G,C2) & (M2=C2) & (M=C) & (M2#M)  % man + cabbage
        )
      &
        % Exclude the forbidden states
        ~(((W=G) | (G=C)) & (M#G))
     )
  )
```

The set of accessible states is computed via fixpoint, and the boatman reaches safely the other bank with his goods since the call of `reachable` succeeds with the four arguments set to `right`:

```
2p |= reachable(Man,Wolf,Goat,Cabbage) ?
```

```
{Man=left, Wolf=left, Goat=left}
{Man=left, Wolf=left, Goat=right, Cabbage=left}
{Man=left, Wolf=right, Goat=left}
{Man=right, Wolf=left, Goat=right}
{Man=right, Wolf=right, Goat=left, Cabbage=right}
{Man=right, Wolf=right, Goat=right}

2p |= reachable(right,right,right,right) ?

1
```

5.2 Nim Game

The Nim game has the following rules:

The game begins with N lines numbered from 1 to N and containing $2 \times i - 1$ matches. At each step, the player who has the turn takes as many matches as he wants in one of the line (but of course, at least one). Then the turn changes. The winner is the player who takes the last match.

In order to model this problem, one first models each line by means of a finite state automaton, itself encoded in Toupie as a ternary relation (initial state, transition label, terminal state). Note that it is not necessary to model the turn explicitely since it changes of value alternatively.

The states of a line represent the number of matches it contains currently. For instance, the line number 3 is modeled by means of a 3 states automaton (0,1,...5 matches). The transition are labelled either with e (the empty transition) or with c. The first one means that the automaton stays in the same state, the second one that the player who has the turn takes a number of matches in this line.

For instance, for the line number 3, it gives:

```
let label {e,c}  % domain of transition labels

let state3 0..5  % domain of state of the line number 3

% The automaton for the line number 3
% S: initial state, L:label, T: terminal state of a transition

line3(S@7:state3,L@8:label,T@9:state3) += (
    ((L=e) & (S=T))
  | ((S=1) & (L=c) & (T=0))
  | ((S=2) & (L=c) & (T in 0..1))
  | ((S=3) & (L=c) & (T in 0..2))
  | ((S=4) & (L=c) & (T in 0..3))
  | ((S=5) & (L=c) & (T in 0..4))
  )
```

In the above program, one has defined the variable indices. The reason is given below.

Now, one models the system compounded by the different lines. This system can be seen as the product of the different automata. The states of the product are the N-tuples of states (a state for each basic automaton) and its transitions are N-tuples of transitions. Of course, all the tuples of transitions are not allowed, since the player who has the turn must take a number of matches in one and only one line. The product is thus synchronized by means of the following predicate (for 3 lines):

```
% one and only one line changes of state at each turn:

synchronizator(L1@2:label,L2@5:label,L3@8:label) += (
```

```
    ((L1=c) & (L2=e) & (L3=e))
  | ((L1=e) & (L2=c) & (L3=e))
  | ((L1=e) & (L2=e) & (L3=c))
  )
```

Now, the reachable states in the synchronized product are computed of two steps: the first one consists in computing all the allowed edges; the second one consists in computing all the states reachable from the initial one, through allowed edges. It has been noted by several authors [Bou93, CGR93, EFT93] that the best variable ordering for computing a synchronized product, is the interleaved one (as defined in the predicate `edge`). This is why this ordering has been choosen for the automata description.

```
edge(S1@1:state1,T1@3:state1,S2@4:state2,T2@6:state2,S3@7:state3,T3@9:state3) +=
  exist L1@2:label, L2@5:label, L3@8:label
    (
        line1(S1,L1,T1)
      & line2(S2,L2,T2)
      & line3(S3,L3,T3)
      & synchronizator(L1,L2,L3)
    )

reachable(T1@3:state1,T2@6:state2,T3@9:state3) += (
    ((T1=1) & (T2=3) & (T3=5))  % initial state
  | exist S1@1:state1, S2@4:state2, S3@7:state3
      (
          reachable(S1,S2,S3)
        & edge(S1,T1,S2,T2,S3,T3)
      )
  )
```

Now, one obtains the configurations where there is a winning strategy by means of the two following predicates:

```
winning(S1@1:state1,S2@4:state2,S3@7:state3) +=
  exist T1@3:state1,T2@6:state2,T3@9:state3
   (
       edge(S1,T1,S2,T2,S3,T3)
     & losing(T1,T2,T3)
   )

losing(S1@1:state1,S2@4:state2,S3@7:state3) +=
  forall T1@3:state1,T2@6:state2,T3@9:state3
   (
        edge(S1,T1,S2,T2,S3,T3)
     => winning(T1,T2,T3)
   )
```

There exists at least one winning strategy for the first player since one obtains:

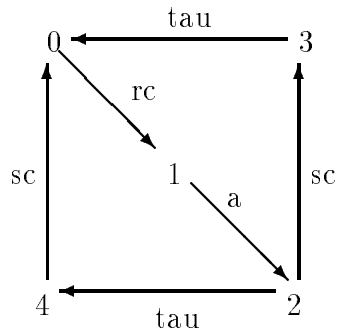
```
2p |= winning(1,3,5) ?

1
```

5.3 Milner's scheduler

This problem is a simple distributed scheduler proposed by Milner [Mil89]. It is now a standard benchmark for process algebra tools [Bou93, EFT93].

The scheduler consists of one starter process and N processes which are scheduled. The communication is organized in a ring. The transition system describing each cycler is depicted below:



Each cycler process C_i awaits the permit (rc) to start, performs the action a, and passes the turn (sc) to the next cycler either before or after some internal computation (tau). The starter just initializes the process.

The system is programmed in the same way that the Nim game is. The cyclers are described by means of the following ternary predicate:

```
let cycler_state 0..4
let cycler_label {e,tau,a,sc,rc}
        % tau : internal action,
        % a   : idem
        % sc  : send the starter to the next cycler
        % rc  : receive the starter from the previous cycler

cycler(S@4:cycler_state,L@5:cycler_label,T@6:cycler_state) += (
    ((L=e) & (S=T))
  | ((S=0) & (L=rc)  & (T=1))
  | ((S=1) & (L=a)   & (T=2))
  | ((S=2) & (L=sc)  & (T=3))
  | ((S=2) & (L=tau) & (T=4))
  | ((S=3) & (L=tau) & (T=0))
  | ((S=4) & (L=sc)  & (T=0))
  )
```

A synchronized product can be computed as in the previous section (we just give the predicate `reachable` for one starter and three cyclers.

```
reachable(TS@3:starter_state,
          T1@6:cycler_state,
          T2@9:cycler_state,
          T3@12:cycler_state) += (
    ((TS=0)&(T1=0)&(T2=0)&(T3=0))
  | exist SS@1:starter_state,
          S1@4:cycler_state,
          S2@7:cycler_state,
          S3@10:cycler_state
      (
```

```
          reachable(SS,S1,S2,S3)
        & edge(SS,TS,S1,T1,S2,T2,S3,T3)
    )
)
```

Now, one can verify that there is no deadlock in the system by means of the following predicate:

```
deadlock(SS@1:starter_state,
         S1@4:cycler_state,
         S2@7:cycler_state,
         S3@10:cycler_state) += (
   reachable(SS,S1,S2,S3)
 & forall TS@3:starter_state,
          T1@6:cycler_state,
          T2@9:cycler_state,
          T3@12:cycler_state
     (
       edge(SS,TS,S1,T1,S2,T2,S3,T3)
       =>
       deadlock(TS,T1,T2,T3)
     )
)
```

Finally one computes a bisimulation on this example that consists of the following steps:

1. Compute the τ-closure of the synchronized product, that is the paths of the form $\tau^\star t \tau^\star$, where t is any transition.

2. Compute the equivalence relation between states using the extended edges above.

We just give hereafter the program for the second step, assuming that the first step has been performed by the predicate `large_edge`.

```
bisimulation(XT@1:starter_state,YT@3:starter_state,
             X1@9:cycler_state,Y1@11:cycler_state,
             X2@13:cycler_state,Y2@15:cycler_state,
             X3@17:cycler_state,Y3@19:cycler_state) -= (
   reachable(XT,X1,X2,X3)
 & reachable(YT,Y1,Y2,Y3)
 & forall UT@2:starter_state,
          U1@10:cycler_state,
          U2@14:cycler_state,
          U3@18:cycler_state
      (
            large_edge(XT,UT,X1,U1,X2,U2,X3,U3)
         => exist VT@4:starter_state,
                  V1@12:cycler_state,
                  V2@16:cycler_state,
                  V3@20:cycler_state
              (
                  large_edge(YT,VT,Y1,V1,Y2,V2,Y3,V3)
                & bisimulation(UT,VT,U1,V1,U2,V2,U3,V3)
              )
      )
 & forall VT@4:starter_state,
          V1@12:cycler_state,
```

```
          V2@16:cycler_state,
          V3@20:cycler_state
      (
            large_edge(YT,VT,Y1,V1,Y2,V2,Y3,V3)
         => exist UT@2:starter_state,
                   U1@10:cycler_state,
                   U2@14:cycler_state,
                   U3@18:cycler_state
              (
                   large_edge(XT,UT,X1,U1,X2,U2,X3,U3)
                 & bisimulation(UT,VT,U1,V1,U2,V2,U3,V3)
              )
      )
 )
```

Bibliography

[Bou93] A. Bouali. *Etudes et mises en œ uvre d'outils de vérification basée sur la bisimulation.* PhD thesis, Université Paris VII, 03 1993. in french.

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*. IEEE 0738, 1990.

[Bry86] R. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 8 1986.

[Bry92] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CCMR93] M-M. Corsini, B. Le Charlier, K. Musumbu, and A. Rauzy. Efficient Abstract Interpretation of Prolog Programs by means of Constraint Solving over Finite Domains (extended abstract). In *Proceedings of the 5th Int. Symposium on Programming Language Implementation and Logic Programming, PLILP'93*, Estonie, 1993.

[CGR93] M-M. Corsini, A. Griffault, and A. Rauzy. Yet another Application for Toupie: Verification of Mutual Exclusion Algorithms. In *proceedings of Logic Programming and Automated Reasonning, LPAR'93*. LNCS, 1993.

[CMR92] M-M. Corsini, K. Musumbu, and A. Rauzy. The μ-calculus over Finite Domains as an Abstract Semantics of Prolog. In M. Billaud, P. Castéran, M-M. Corsini, K. Musumbu, and A. Rauzy, editors, *WSA'92 Workshop on Static Analysis (Bordeaux)*, volume 81–82 of *Bigre*, pages 51–59. Atelier Irisa, Sept. 23–25 1992.

[CR93a] M-M. Corsini and A. Rauzy. First Experiments with Toupie. Technical Report 577–93, LaBRI - Université Bordeaux I, 1993.

[CR93b] M-M. Corsini and A. Rauzy. Toupie: the μ-calculus as a Constraint Language, 1993. submitted to the Journal of Automated Reasonning.

[EFT93] R. Enders, T. Filkorn, and D. Taubner. Generating bdds for symbolic model checking in ccs. *Journal of Distributed Computing*, 6:155–164, 6 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

Appendix A

Syntax

This appendix recalls the syntax of Toupie programs and requests in Bacchus Normal Form.

```
<request> ::= <fixed-point-or-request>
          ::= <formula> ?

<fixed-point-or-request> ::= <predicate> += <formula>
                         ::= <predicate> -= <formula>
                         ::= <predicate> ?

<formula> ::= <parenthesed-formula>
          ::= <projection>
          ::= <cardinality>
          ::= <at-least>
          ::= <quantifier>
          ::= <not>
          ::= <boolean-constant>
          ::= <predicate>

parenthesed-formula> ::= ( <atomic-relation> )
                     ::= ( <composed-formula> )

<composed-formula> ::= <n-ary-connective>
                   ::= <binary-connective>
                   ::= <formula>

<binary-connective> ::= <formula> => <formula>
                    ::= <formula> <=> <formula>
                    ::= <formula> <#> <formula>

<n-ary-connective> ::= <item-list[ <formula>  | ]>
                   ::= <item-list[ <formula>  & ]>

<predicate> ::= <identifier> ( item-list[ <argument> , ] )

<not> ::= ~ <formula>

<boolean-constant> ::= 0 | 1

<atomic-relation> ::= <variable> = <variable>
```

```
                ::= <variable> # <variable>
                ::= <variable> = <constant>
                ::= <variable> # <constant>
                ::= <variable> in <domain>

<quantifier> ::= exist <variables-list> <formula>
             ::= forall <variables-list> <formula>

<at-least> ::= @(<integer>, [ <item-list[ <formula> , ]> ])

<cardinality> ::= #(<integer>, <maximum>, [<item-list[<formula> ,]>])

<projection> ::= [ <formula>; <assignment-list> ]

<assignment-list> ::= <variable> <- <argument> , <assignment-list>
                  ::= <variable> <- <argument>

<item-list[ <item> <separator> ]>
           ::= <item> <separator> <item-list[ <item> <separator> ]>
           ::= <item>

<argument> ::= <constant> | <domain> | <variable>

<domain> ::= <constant-set> | <range> | <functor>

<constant-set> ::= { <item-list [ <constant> , ]> }

<range> ::= <integer>..<integer>

<constant> ::= <integer> | <identifier>

<identifier> ::= any string of alphanum or _ letters beginning with
                 a lower case letter.

<variable> ::= any string of alphanum or _ letters beginning with
               a lower case letter.
```

Appendix B

Reported Bugs

This appendix contains some bugs fixed in the version 0.24.

- If a constant is repeated in a domain definition, no error nor warning message is printed. E.g.

```
2p |= let a_domain {a, a, a}

2p |= (X:a_domain = Y: a_domain) ?

{X=a, Y=a}
{X=a, Y=a}
{X=a, Y=a}
```

- If a fixpoint equation is defined twice, the two versions are maintained. It may result an unexpected behavior.

```
2p |= let dom {a,b}

2p |= p(X:dom,Y:dom) += (X=Y)

2p |= p(X:dom,Y:dom) += (X#Y)

2p |= p(X:dom,Y:dom) ?

{X=a, Y=a}
{X=b, Y=b}
```

- A variable cannot be declared twice in a request or a fixpoint definition, even if it is declared under the scope of a quantifier.

```
2p |= p(X:{a,b},Y:1..2) += (
          exist Z:{a,b} (X=Z)
        & exist Z:1..2  (Y=Z)

2p ->  line 3: the variable Z has been used with another domain.
```